2 OPERATING SYSTEM KERNEL CONCEPTS

In This Chapter

This chapter concentrates on concepts, motivation, and general architectural principals of the operating system kernel. It also provides information on how to partition a VDK application into independent, reusable functional units that are easy to maintain and debug.

The following sections provide information about the operating system kernel concepts:

- "Motivation" on page 2-2
- "Partitioning an Application" on page 2-5
- "Scheduling" on page 2-6
- "Protected Regions" on page 2-9
- "Thread and Hardware Interaction" on page 2-11

Motivation

All applications require control code as support for the algorithms that are often thought of as the "real" program. The algorithms require that data being moved to and/or from peripherals, and many algorithms consist of more than one functional block. For some systems, this control code may be as simple as a "superloop" blindly processing data that arrives at a constant rate. However, as processors become more powerful, considerably more sophisticated control may be needed to realize the processor's potential, to allow the DSP to absorb the required functionality of previously supported chips, and to allow a single DSP to do the work of many. The following sections provide an overview of some of the benefits of using a kernel on a DSP.

Rapid Application Development

The tight integration between the VisualDSP++ environment and the VDK allows rapid development of applications compared to creating all of the control code required by hand. The use of automatic code generation and file templates, as well as a standard programming interface to device drivers, allows you to concentrate on the algorithms and the desired control flow rather than on the implementation details. VDK supports the use of C, C++, as well as assembly language. You are encouraged to develop code that is highly readable and maintainable, yet to retain the option of hand optimizing if necessary. For more information, see "Configuring a Project" on page 3-2.

Debugged Control Structures

Debugging a traditional DSP application can be laborious because development tools (compiler, assembler, and linker among others) are not aware of the architecture of the target application and the flow of control that results. Debugging complex applications is much easier when instantaneous snapshots of the system state and statistical run-time data are

clearly presented by the tools. To help offset the difficulties in debugging software, VisualDSP++ includes instrumented and non-instrumented versions of the VDK libraries.

In the instrumented mode, the kernel maintains statistical information and logging of all significant events into a history buffer. When the execution is paused, the debugger can traverse this buffer and present a graphical trace of the program's execution including context switches, pending and posting of signals, changes in a thread's status, and more. Statistics are presented for each thread in a tabular view and show the total amount of time the thread has executed, the number of times it has been run, the signal it is currently blocked on, and other data. For more information, see "Debugging a VDK Project" on page 3-44.

Code Reuse

Many programmers begin a new project by writing the infrastructure portions that transfers data to, from, and between algorithms. This necessary control logic usually is created from scratch by each design team and infrequently reused on subsequent projects. The VDK provides much of this functionality in a standard, portable and reusable library. Furthermore, the kernel and its tight integration with the VisualDSP++ environment are designed to promote good coding practice and organization by partitioning large applications into maintainable and comprehensible blocks. By isolating the functionality of subsystems, the kernel helps to prevent the morass all too commonly found in systems programming.

The kernel is designed specifically to take advantage of commonality in user applications and to encourage code reuse. Each thread of execution is created from a user-defined template, either at boot time or dynamically by another thread. Multiple threads can be created from the same template, but the state associated with each created instance of the thread remains unique. Each thread template represents a complete encapsulation of an algorithm that is unaware of other threads in the system unless it has a direct dependency.

Hardware Abstraction

In addition to a structured model for algorithms, the VDK provides a hardware abstraction layer. Presented programming interfaces allow you to write most of the application in a platform-independent, high-level language (C or C++). The VDK API is identical for all Analog Devices processors, allowing code to be easily ported to a different DSP core.

When porting an application to a new platform, programmers must address the two areas that are necessarily specific to a particular processor: interrupt service routines and device drivers. The VDK architecture identifies a crisp boundary around these subsystems and supports the traditionally difficult development with a clear programming framework and code generation. Both interrupts and device drivers are declared with a graphical user interface in the IDDE, which generates well-commented code that can be compiled without further effort.

Partitioning an Application

A VDK thread is an encapsulation of an algorithm and its associated data. When beginning a new project, use this notion of a thread to leverage the kernel architecture and to reduce the complexity of your system. Since many algorithms may be thought of as being composed of "sub-algorithm" building blocks, an application can be partitioned into smaller functional units that can be individually coded and tested. These building blocks then become reusable components in more robust and scalable systems.

You define the behavior of VDK threads by creating thread types. Thread types are templates that define the behavior and data associated with all threads of that type. Like data types in C or C++, thread types are not used directly until an instance of the type is created. Many threads of the same thread type can be created, but for each thread type, only one copy of the code is linked into the executable code. Each thread has its own private set of variables defined for the thread type, its own stack, and its own C run-time context.

When partitioning an application into threads, you should identify portions of your design in which a similar algorithm is applied to multiple sets of data. These are, in general, good candidates for thread types. When data is present in the system in sequential blocks, only one instance of the thread type is required. If the same operation is performed on separate sets of data simultaneously, multiple threads of the same type can coexist and be scheduled for prioritized execution (based on when the results are needed).

Scheduling

The VDK is a pre-emptive multitasking kernel. Each thread begins execution at its entry point. Then, it either runs to completion or performs its primary function repeatedly in an infinite loop. It is the role of the scheduler to pre-empt execution of a thread and to resume its execution when appropriate. Each thread is given a priority to assist the scheduler in determining precedence of threads.

The scheduler gives processor time to the thread with the highest-priority that is in the *ready* state (see Figure 4-2 on page 4-14). A thread is in the ready state when it is not waiting for any system resources that it has requested, and a reference to each ready thread is stored in a structure internal to the kernel known as the *ready queue*. For more information, see "Scheduling" on page 4-9.

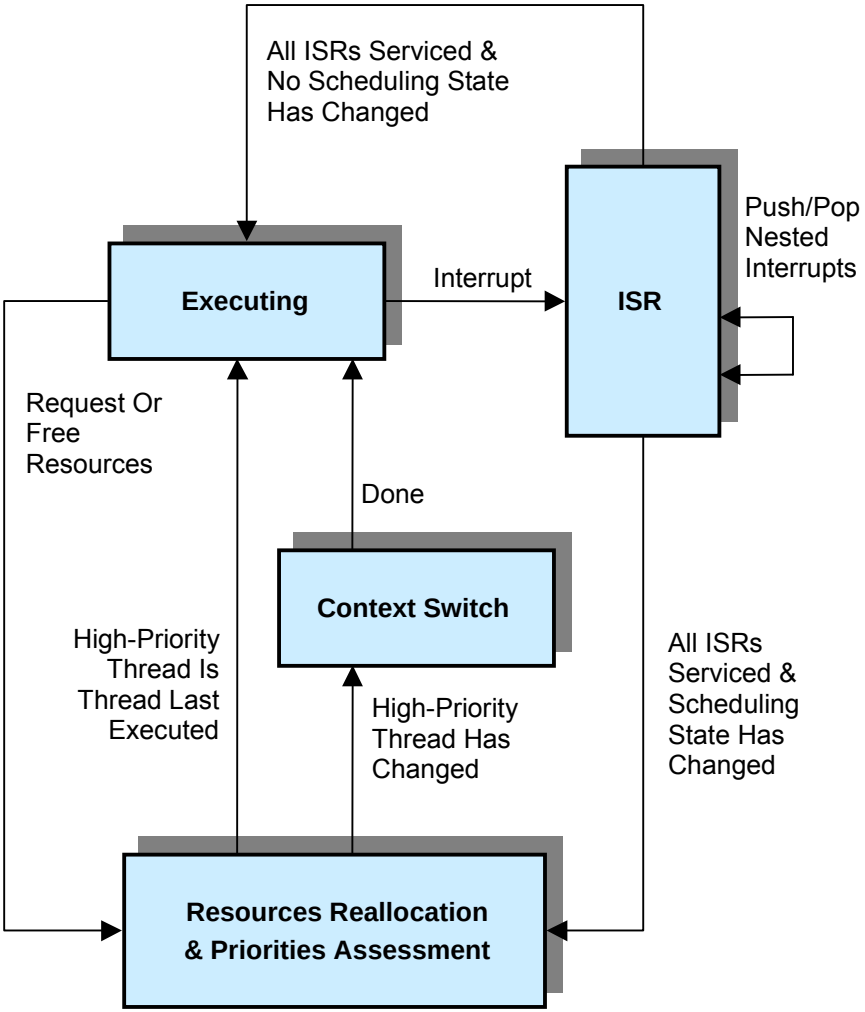

Figure 2-1. VDK State Diagram

Priorities

Each thread is assigned a dynamically-modifiable priority based on the default for its thread type declared in VisualDSP++ environment's **Project** window. An application is limited to either fourteen or thirty priority levels, depending on the processor's architecture. However, the number of threads at each priority is limited, in practice, only by system memory. Priority level one is the highest priority, and priority fourteen (or thirty) is the lowest. Note that the system maintains an idle thread that is set to a priority lower than that of the lowest user thread.

Assigning priorities is one of the most difficult tasks of designing a real-time pre-emptive system. Although there has been research in the area of rigorous algorithms for assigning priorities based on deadlines (e.g., rate monotonic scheduling), most systems are designed by considering the interrupts and signals that are triggering the execution, while balancing the deadlines imposed by the system's inputs and output streams. For more information, see "Thread Parameters" on page 4-2.

Pre-emption

A running thread continues execution unless it requests a system resource using a kernel API. When a thread requests a signal (semaphore, event, or device flag) and the signal is available, the thread resumes execution. If the signal is not available, the thread is removed from the ready queue—the thread is *blocked* (see Figure 4-2 on page 4-14). Note that the kernel does not perform a context switch as long as the running thread maintains the highest priority in the ready queue, even if the thread frees a resource and enables other threads to move to the ready queue at the same or lower priority. A thread can also be interrupted. When an *interrupt* occurs, the kernel yields to the hardware interrupt controller. When the ISR completes, the highest-priority thread resumes execution. For more information, see "Pre-Emptive Scheduling" on page 4-11.

Protected Regions

Frequently, system resources must be accessed atomically. The kernel provides two levels of protection for code that needs to execute sequentially—unscheduled regions and critical regions.

Critical and unscheduled regions can be intertwined. You can enter critical regions from within unscheduled regions, or enter unscheduled regions from within critical regions. For example, if you are in an unscheduled region and call a function that pushes and pops a critical region, the system is still in an unscheduled region when the function returns.

Disabling Scheduling

The VDK scheduler can be disabled by entering an *unscheduled region*. The ability to disable scheduling is necessary when you need to free multiple system resources without being switched out, or access global variables that are modified by other threads without preventing interrupts from being serviced. While in an unscheduled region, interrupts are still enabled and ISRs execute. However, the kernel does not perform a thread context switch even if a higher-priority thread becomes ready. Unscheduled regions are implemented using a stack style interface. This enables you to begin and end an unscheduled region within a function without concern for whether or not the calling code is already in an unscheduled region.

Disabling Interrupts

On occasions, disabling the scheduler does not provide enough protection to keep a block of thread code reentrant. A *critical region* disables both scheduling and interrupts. Critical regions are necessary when a thread is modifying global variables that may also be modified by an Interrupt Service Routine. Similar to unscheduled regions, critical regions are implemented as a stack. Developers can enter and exit critical regions in a

function without being concerned about the critical region state of the calling code. Care should be taken to keep critical regions as short as possible as they increase interrupt latency.

Thread and Hardware Interaction

Threads should have minimal knowledge of hardware; rather, they should use *device drivers* for hardware control. A thread can control and interact with a device in a portable and hardware abstracted manner through a standard set of APIs.

The VDK Interrupt Service Routine framework encourages you to remove specific knowledge of hardware from the algorithms encapsulated in threads (see Figure 2-2). Interrupts relay information to threads through signals to device drivers or directly to threads. Using signals to connect hardware to the algorithms allows the kernel to schedule threads based on asynchronous events.

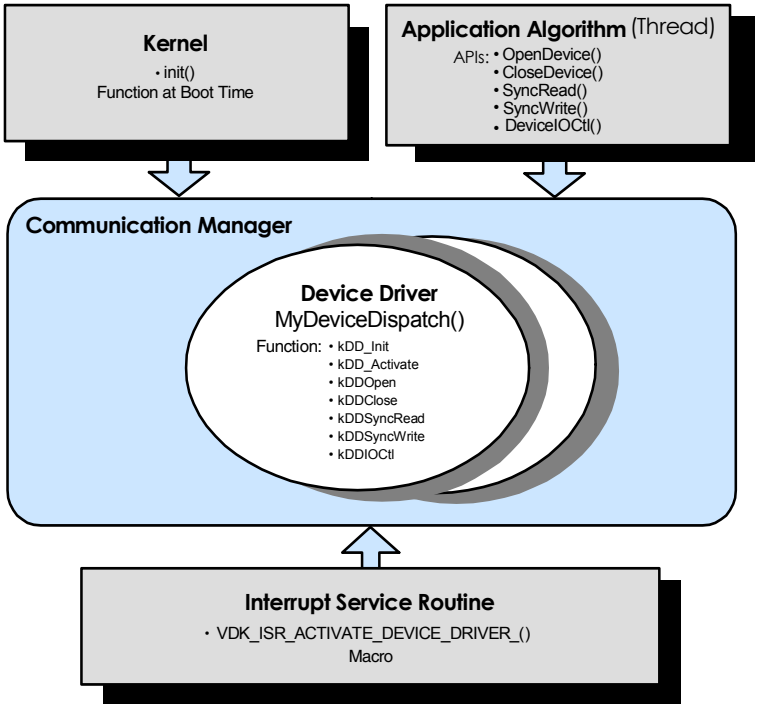

Figure 2-2. Device Drivers' Entry Points

The VDK run-time environment can be thought of as a bridge between two domains, the thread domain and the interrupt domain. The interrupt domain services the hardware with minimal knowledge of the algorithms, and the thread domain is abstracted from the details of the hardware. Device drivers and signals bridge the two domains. For more information, see “Threads” on page 4-2.

Thread Domain with Software Scheduling

The thread domain runs under a C/C++ run-time model. The prioritized execution is maintained by a software scheduler with full context switching. Threads should have little or no direct knowledge of the hardware; rather, threads should request resources and then wait for them to become available. Threads are granted processor time based on their priority and requested resources. Threads should minimize time spent in critical and unscheduled regions to avoid short-circuiting the scheduler and interrupt controller.

Interrupt Domain with Hardware Scheduling

The interrupt domain runs outside the C/C++ run-time model. The prioritized execution is maintained by the hardware interrupt controller. ISRs should be as small as possible. They should only do as much work as is necessary to acknowledge asynchronous external events and to allow peripherals to continue operations in parallel with the processor. ISRs should only signal that more processing can occur and leave the processing to threads. For more information, see “Interrupt Service Routines” on page 4-30.

Device Drivers

Interrupt Service Routines can communicate with threads directly using signals. Alternatively, an Interrupt Service Routine and a thread can use a device driver to provide more complex device-specific functionality that is abstracted from the algorithm. A device driver is a single function with multiple entry conditions and domains of execution. For more information, see “Device Drivers” on page 4-35.